SPaCIoS

Secure Provision and Consumption in the Internet of Services

FP7-ICT-2009-5, ICT-2009.1.4 (Trustworthy ICT)

Project No. 257876

www.spacios.eu

Deliverable D2.1.3

Analysis of the relevant concepts used in the new case studies: applicable security concepts, security goals and attack behaviors

Abstract

This deliverable discusses applicable security concepts (modeling aspects, goals and attack behaviors) for the already defined case studies of the SPaCIoS project and for the newly added ones, from an open-source perspective. It also presents the approach for extracting models from source code, with examples of manually constructed models anticipating their automatic generation.

Deliverable details

Deliverable version: *v1.0*
Classification: *public*
Date of delivery: *31.03.2012*
Due on: *31.03.2012*
Editors: *IeAT, UNIVR*
Total pages: *28*

Project details

Start date: *October 01, 2010*
Duration: *36 months*
Project Coordinator: *Luca Viganò*
Partners: UNIVR, ETH Zurich, INP, KIT, UNIGE, SAP, Siemens, IeAT

SEVENTH FRAMEWORK PROGRAMME

(this page intentionally left blank)

Contents

List of Figures

1 Introduction

Models are the central artifact of the SPaCIoS project. They can be written manually or learned by observing system executions. A new focus brought by the extension of the SPaCIoS with the partner IeAT is the extraction of models from source code.

An initially planned case study in this respect was the mOSAIC open-source cloud computing platform. As the security-related functionality in the mOSAIC project is not yet developed, analysis of this case study had to be postponed, together with the decision on its suitability and feasibility for the SPaCIoS project.

As an alternative, we have identified a set of open-source applications originally developed by Yes Software as part of a portal for web developers, under the name *GotoCode* [11]. These applications (including an online bookstore, a site for classifieds, an employee directory, etc.) have already been used in the research literature as targets for both source code analysis and vulnerability testing. They are medium-size applications, making them suitable as targets for developing and testing our analyses. Like the Webgoat case study, the GotoCode applications are written for very common usage scenarios. They do not contain intentionally injected flaws, but have confirmed vulnerabilities which make them interesting as analysis targets.

In addition to analyzing the security concepts relevant for extracting models from the source code of these new case studies and others already targeted by SPaCIoS, a major part of this deliverable presents an overview of the proposed model extraction process. We focus on the abstraction process involved in going from Java source code to ASLan models, and describe initial results for constructing a graph model for the structure and behavior of a representative class of applications using Java Server Pages (JSP) and Java Servlets. We illustrate the description with examples of hand-written and automatically constructed models.

2 Open-source case studies

2.1 GotoCode applications

In order to test our approach of extracting models from source code, we have selected a suite of test cases that are relevant to the goal of finding security issues, and were used for similar purposes in the literature. Therefore, in addition to the systems SPaCIoS already considered, we have started analyzing the *GotoCode* Web applications [11], which we have found interesting because they contain very typical every-day application scenarios.

GotoCode is a set of JSP-based programs that implement several typical applications such as: a bookstore, a forum, a directory, etc. They were used in literature for different analysis purposes, e.g., from software testing [3, 5], failure detection [9], to vulnerability detection [3, 10, 4].

The list of GotoCode applications includes:

- an online bookstore, with registration, product voting, and administration
- a bug tracking system, with authentication, project and user administration
- a platform for classifieds
- an employee directory with search, member and department administration
- an event management system with add, search and administration
- an online discussion forum with threads and keyword search
- a web-based ledger that can track deposits and withdrawals
- a link management system with administration and approval
- an online portal with registration, link approval and administration
- a registration form with field validation
- a task manager with project, person and priority administration
- a yellow pages application with administration of listings, users and categories

The applications range in size from about a dozen classes for the smaller ones to 28 classes for the largest (bookstore and portal), which have about 10 000 lines of JSP code.

There are several advantages in analyzing the GotoCode applications. Their size is adequate for testing an analysis approach, as they provide a small

enough yet highly relevant input for any automated tool that analyzes the source code to extract information about possible vulnerabilities. Moreover, the code is very cleanly structured. This makes program comprehension fairly straightforward, which enables us to easily verify the output of our techniques or tools.

2.2 Existing SPaCIoS case studies

WebGoat [7] is a Web application developed with the specific purpose of studying common security-related vulnerabilities in real-world programs. It consists of a set of lessons structured around particular Web application examples, each describing a specific attack. The examples provide the vulnerable code, on which the user can exercise the most common paths of attack, by following the needed steps that enact the vulnerability. The applications are built using the Jakarta ECS Web application framework to generate the dynamic HTML content to the client.

The major advantage of using WebGoat as a case study is that, besides being easy to understand at the source level, it focuses on clearly specified vulnerabilities in the analyzed code, which helps both the goal of modeling the system starting with its source code, and the validation of the approach.

The other open-source case studies of SPaCIoS: OAuth, OpenID and SAML SSO, have an increased level of complexity. The usability of the developed model extraction techniques will be evaluated in a second stage, after the implementation has matured through the lessons learned on GotoCode and WebGoat. Priority will be given to SAML SSO, since the significant related expertise in the consortium offers the best potential to leverage a combination of modeling techniques.

The implementation of model extraction will be targeted towards Java applications, and initially towards JSP/Servlets. However, one of the most widely used implementations, simpleSAMLphp, is written in PHP. A static analysis infrastructure for PHP, `pfff`, has been developed by Facebook and is available open-source, written in OCaml. If the developed model extraction is sufficiently generic, we will pursue this direction to make it available for more than one language and platform.

3 Analysis of relevant security concepts

We follow the structure of Deliverable D2.1.1 [8] which has presented the various security aspects relevant for the SPaCIoS application scenarios and the extent to which they can be modeled in ASLan++ or ASLan. We discuss the relevance of these aspects to the newly introduced GotoCode case studies, as well as the extent to which they can be handled in models extracted from source code.

3.1 Security aspects

Seven categories of security-relevant aspects were identified in D2.1.1:

Input and output validation / sanitization This category is of concern to practically any application. In particular, most of the GotoCode applications operate with SQL databases, creating queries starting from its inputs, which would need to be sanitized. However, as the ASLan++ / ASLan languages is not suitable for modeling this sanitization, the model extraction process will abstract away from these issues. As per D2.1.1, they fall within the scope of the vulnerability-driven testing module. Static analysis tools for taint checking are outside the scope of this work.

Error handling Precise modeling of error handling is essential for confidence in the verification result, since getting to an unpredicted state after handling an error may be the entry point for an attack. The models extracted from application source code will model the control paths for error handling, including exceptions. If the resulting models are impractical to verify due to the state explosion problem, we will consider an approach that generates a partial model for the normal control path, whereas checking for the error path is done separately or relegated to testing.

Channel protection All other application scenarios in SPaCIoS rely on secure channels. This is not explicitly the case in the GotoCode applications, thus by default any communication is visible to the intruder. However, extracting a model of the application logic from its source code is independent of the communication channels. Thus, the resulting ASLan model can be adapted to use secure channels and then check if any vulnerabilities are still present after this change.

Modularity, concurrency and control flows The GotoCode case studies model the server side of an application; it should be able to receive concurrent requests from any number of clients. Moreover, although there may be an intended sequence of events, reflected in the sequence of web pages (including forms) presented to the client, the actual behavior is not restricted to this sequence, as a (possibly malicious) client may perform arbitrary requests in any order. This will be reflected in the generated models, which, like the code, have the structure of a server loop that accepts requests and processes them based on their parameter values and the stored local state.

However, the extracted models will not represent any part of the HTTP server infrastructure, which is assumed to function correctly according to specification. Thus, the models will not account for any potential flaws in the HTTP server that may be triggered in conjunction with the application.

Cryptographic operations There are no cryptographic operations explicitly represented in the GotoCode applications. As in the entire SPaCIoS project, we will assume cryptographic operations to be abstract and perfect. In general, to generate ASLan models that employ cryptographic primitives, the model extraction procedure will need to be provided with a mapping between the language- and framework-specific API to such primitives and their ASLan representation.

Cookies and authentication tokens As expected, the GotoCode applications rely on session state to keep track of various elements, such as whether a user has been authenticated, or the setting of the user's privilege level. This server state will be modeled using state variables in ASLan++ and/or state facts in ASLan.

Authorization policies and access control GotoCode applications do not come with specifications of authorization policies or any other kind. In general, model extraction will be independent of such specifications, as there is no standard way to represent them in source code. However, practically in all applications, some level of authorization is implicit. Almost all applications have administration features, and the code contains checks that the features can only be accessed from a certain privilege level. These can serve as the basis for writing the corresponding specification, although we do not envision deriving them automatically. As the authorization rules are simple, it may not be needed to model them as Horn clauses.

3.2 Security goals

Although none of the applications comes with explicit security requirements, these arise naturally from their functionality, allowing sensible and representative security goals to be specified. Following the approach of D2.1.1, they can be classified as follows:

Confidentiality As perhaps the most widespread security goal, confidentiality is desirable in several of the GotoCode applications. For example, the online bookstore uses a shopping cart, the contents of which should not be accessible to an intruder; the same goes for the bank balance in the ledger application, etc. Obviously, regardless of the model extracted for the application logic, communication over a secure channel is necessary (but not sufficient) for confidentiality, and it is under this assumption that confidentiality specifications will be checked.

Privacy Given a notion of ownership, we can in addition specify privacy goals. This can be done for the application scenarios described under confidentiality above (e.g., the shopping cart should not be accessible to a different registered user), and others, e.g., in the employee directory, (parts of) a profile should not be accessible to other users.

Authentication and authenticity Most GotoCode applications require users to authenticate, by using a login name and a password, which are checked against a database. The database itself will be abstracted away in the model, and the check replaced with a boolean function. Similar to other SPaCIoS case studies, we will need to model that the intruder cannot generate a valid username–password combination.

Integrity Several of the GotoCode applications have integrity constraints: for the bookstore, the user's shopping cart should not be modified by anyone else, similarly for the ledger, or the records in the employee directory.

Authorization and access control Most GotoCode applications come with administration facilities, for users, departments, events, projects, etc. The user access level is retrieved from the database at login time. Subsequent checks against this level are made on each attempt at administrative access, allowing authorization specifications to be written based on these checks.

Based on our initial assessment, there are no provisions or requirements for accountability, non-repudiation, or availability in the selected case studies.

3.3 Vulnerabilities

From the OWASP vulnerability categories discussed in D2.1.1, the following are relevant for the new GotoCode application scenarios:

Authentication vulnerabilities Issues in this category include compromising user credentials by transmitting passwords over an insecure channel, or implementation errors (no deauthentication of the user upon logout, confusion upon login of different users, etc).

Authorization vulnerabilities The functionality of GotoCode applications is typically available only to authorized users upon authentication. Trying to buy a book in the online bookstore for instance will redirect the user to the login page. However, it is possible that such checks have been omitted in places, or that an unauthorized user could simply access a resource through a known link, without going through the presentation layer.

Concurrency vulnerabilities These may appear, e.g., due to invalid use of static fields, or other interactions involving shared server state. By default, all models will be subject to model checking using multiple concurrent sessions.

Error handling vulnerabilities Vulnerabilities such as an entry point without authentication can occur due to an inconsistent or unexpected (unintended) state after error processing can be triggered. An important problem in detecting such vulnerabilities is the precise modeling of exception handling in the Java code of the applications.

Session management vulnerabilities Most GotoCode applications are vulnerable to session riding (CSRF) attacks, since requests do not include any session-specific token, thus, after authentication, a malicious request through a link within the same browser session will be accepted and processed. This presents us with a first known vulnerability to be reproduced using the extracted models.

We have not identified relevant configuration capabilities in the GotoCode applications, and we do not model vulnerabilities due to various server or network (mis)configurations, or protocol errors. Our initial analysis did not find direct uses of path names that would open up relevant vulnerabilities. Low-level vulnerabilities due to missing or incorrect input validation are outside the scope of the model-checking tools and will be handled by testing.

3.4 Attacker behavior and models

Of the CAPEC attack patterns identified in D2.1.1, we consider the following to be particularly relevant for the GotoCode applications:

Forceful browsing (CAPEC-87)
An intruder may bypass the presentation layer and directly perform any requests based on the previously observed structure and parameters of the web application. An exhaustive check against such attacks is a major benefit of extracting a complete model from the application source code.

Exploitation of session variables, resource IDs and other trusted credentials (CAPEC-21)
Potentially, because of coding errors, an application might accept an unauthentic request made on behalf of a user. Again, constructing a precise model will allow to detect or disprove such attacks.

Cross site request forgery (session riding) (CAPEC-62)
As discussed previously, the GotoCode applications are actually vulnerable to such resource integrity attacks, which we expect to confirm using the constructed models.

4 Model extraction approach

We present an initial outline of our approach to extracting ASLan models from Java applications. We focus on a representative class of applications, namely those written using Java Server Pages (JSP) and servlets. We first discuss the overall abstraction process used to obtain an ASLan mode, illustrated with an example. Then we present a prototype model builder that automates the first phase of this process, extracting the graph structure of the dynamically generated HTML pages of such an application.

4.1 Technological Aspects

Several specific challenges must be taken into consideration when designing a technique to automatically extract a behavioral model from source code of distributed, service-oriented applications.

First of all, service-oriented applications crucially depend on the technological aspect. The components of a distributed software system rely on a set of services that allow them to communicate over the network, coordinate their activities, and generally create the environment they need at runtime. The software infrastructures that provide these kinds of services define specific constraints on the design and implementation of the applications. For instance, using Java RMI to enable objects to communicate over the network forces the developers to implement certain interfaces, to use specific library calls for connecting to the remote objects or to the naming service, and so on. While for technologies like RMI the constraints are minimal and only influence the application at the implementation level, other types of environments define rules and constraints that even influence the design. Examples include EJB, JMS, and the plethora of Web application frameworks.

The constraints imposed by the technology have two major consequences for a reverse engineering a formal behavioral model from the implementation:

1. Using third-party technologies adds to the application a level of complexity that usually makes the analysis harder. For example, depending on its design quality, an application may inadvertently mix high-level concerns related to its specification-derived goals with the low-level functionality that deals with communication or other technology-related aspects. Such an application places additional burdens on the analysis, requiring it to properly extract or separate the semantically distinct concerns. Moreover, because of the variety of available technologies, a single approach cannot be developed to cover them all.

2. On the other hand, if properly understood, the technological constraints may become a key asset. Detecting the various known code patterns that are specific to a certain technological infrastructure can lead to deriving relevant aspects for the analysis goals, such as finding the component roles (e.g., client, server), finding the code fragments that build the component's response to its users (e.g. those that use sockets to send data), capture the aspects related to the inter-component communication protocols, and so on.

Another important issue to consider relates to the inherent diversity of the technologies used by distributed applications. The multitude of libraries, frameworks, application servers, and other types of environments available to the developers create a very complex analysis context. Each technology uses specific concepts and imposes constraints that are very different from those defined by most of the others. The task of reverse engineering becomes difficult, and forces the researcher to focus its attention on particular types of technologies, and indeed to specific patterns of their usage, rather than trying to describe a markedly elusive "general" analysis approach. In analyzing relevant web applications we have found that the very wide variety of choices to design and implement such systems allows the developers to freely combine various types of technologies in the same application, even though those technologies may seem distinct or even incompatible to each other.

For our goal of reverse engineering behavioral models of service-oriented applications from their source code, we have focused on web applications mainly because of their popularity. As noted above, there is a wide variety of techniques for developing such applications, which may be freely combined in the same system. To properly channel our approach, we have decided to analyze systems built using two related technologies: *Java Server Pages (JSP)* and *Java Servlets*, both parts of the Java2 Enterprise Edition platform.

4.1.1 The structure of a JSP/Servlets application

As all web applications, a JSP/Servlets application is a predominantly server-side software system. The clients, general-purpose web browsers, only run limited system functionality, through specific technologies such as Javascript or web browser plugins. The server runs in an environment provided by a technology-specific application server (e.g., Tomcat) that hosts the main part of the application, and can optionally be connected to a "classic" web server. The communication between the client and the server is done using the HTTP protocol. Clients (the browsers) generate requests, and the server answers by providing them static or dynamically generated HTML pages, which are then presented by the browsers to the users.

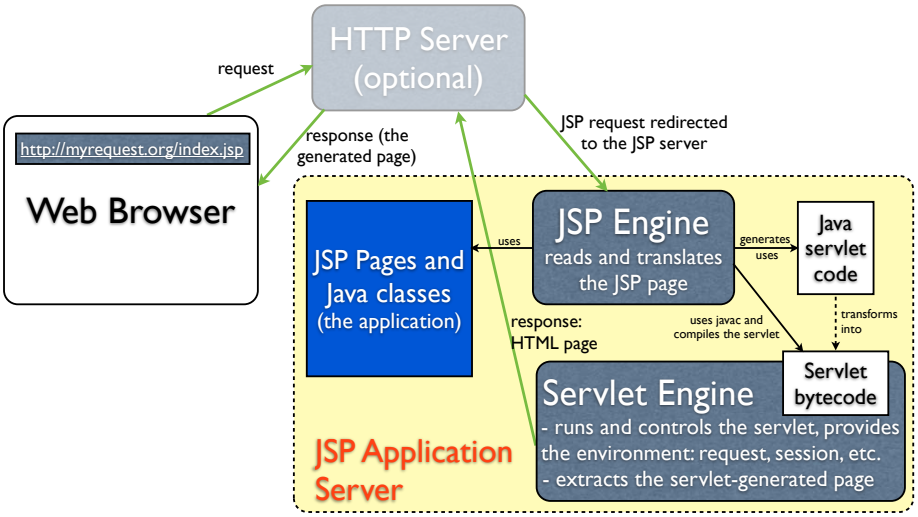

Figure 1: The anatomy of a JSP application

To dynamically generate web pages, developers need to implement Java classes known as *servlets*, or write higher-level "JSP pages", and then deploy them within the application server's environment. A servlet is a Java class that implements specific methods and is written according to a set of rules defined by the Java Servlets technology. A JSP page is a text page residing on a server, and mixes HTML with Java code to describe the layout of the page that has to be generated and sent to the browser.

The constraints these technologies impose on design and implementation can provide valuable information to an analysis that extracts a behavioral model. JSP pages and specific statements in servlets describe the layout of the web pages that are generated on client request, providing information about the various types of data exchanged with the user. For example, the forms included in a page define the fields the user will fill in at the next step, as well as other attributes (e.g., hidden fields) that will be sent in the subsequent HTTP request from the browser. This information can be used to extract the content of the messages exchanged between the two parties. Links that are generated in the page provide two types of information: attributes that show up in the next request, and references to other server-side components (web pages) that can be reached from the current component. Our approach uses all this information about the structure of the communication between the client and the server to guide the model extraction process.

Servlets and JSP pages can be freely mixed within an application. Moreover, the two technologies are directly related, in that that JSP pages are

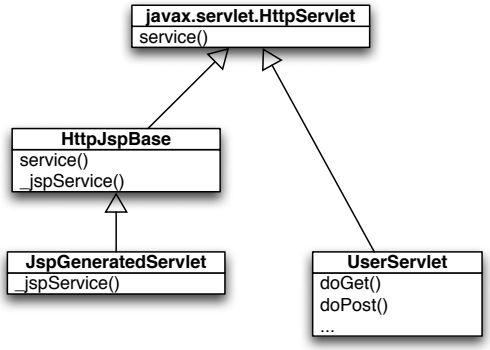

Figure 2: User servlets and JSP-generated servlets

actually transformed by the application server, at runtime, into servlet classes that generate the dynamic pages. The process is depicted in Figure 1, which also shows the general structure of a web application.

4.1.2 Generating servlets from JSP

For a consistent approach to both JSP and servlet applications, we start by automatically transforming all JSP pages in the application into servlets (as Java source code), then we analyze the servlets themselves. For this purpose, we use the native JSP compiler (`jspc`) found in Apache Tomcat JSP Engine (Jasper 2) to generate the servlets.

While the resulting Java classes are servlets, they are not directly derived from `HTTPServlet`, unlike a normal servlet (Figure 1). Instead, their base class is a specific Tomcat class: `org.apache.jasper.runtime.HttpJspBase`. However, this does not complicate our program analysis. `HttpJspBase` is a simple class that directly extends `HTTPServlet`, only performing some initialization and destruction specific to the servlet framework, and overriding the `service()` method of the generic servlet by calling a newly defined abstract `_jspService()` method.

As a consequence, `_jspService()` will be present in all servlets generated by `jspc` for the input JSP pages. It is the unique entry point in the servlet, and receives two arguments: the HTTP request and the HTTP response.

The code in the generated `_jspService()` closely follows the JSP page: the HTML is generated by `out.print()` / `out.println()` statements, while the Java code is properly inserted at the correct locations. The JSP-dependent generated code is easy to find within the `try` ... `catch` block, it is always located just after a generic (and constant) initialization code.

JSP-generated servlets differ from the usual servlets written without JSP

in their implementation of the `service()` method. Specifically, `HttpJspBase` overrides the protected `service()` method in `HTTPServlet`. In a normal servlet, this method processes the HTTP request and dispatches the execution to one of the `doGet`, `doPost`, `doDelete`, etc. methods, according to the HTTP command received. `HttpJspBase` changes this behavior, and simply calls the abstract `_jspService()` of the generated servlets. Thus, servlets generated from JSP have unique entry points, while an analysis for normal servlets must consider potentially multiple `doXY()` entry points, with specific purpose depending on the type of operation: post, delete, put, etc. However, this distinction is easy to deal with in a support tool, and does not add any complications to the analysis approach itself.

4.2 Abstracting JSP applications into ASLan models

We will outline the model extraction process using an example from the GotoCode bookstore application, written using the Java Server Pages technology. As described in the previous section, the resulting servlet code has a unique entry point, the `_jspService` method, with two arguments, the HTTP request and response.

```
public void _jspService(
  final javax.servlet.http.HttpServletRequest request,
  final javax.servlet.http.HttpServletResponse response)
  throws
    java.io.IOException, javax.servlet.ServletException {
...
  javax.servlet.http.HttpSession session = null;
  javax.servlet.jsp.JspWriter out = null;
```

The method manipulates an `HttpSession session` object which represents the session state, and a `JspWriter out` object representing the output stream produced by the servlet, i.e., the generated HTML.

Thus, the resulting ASLan model[1] will have a loop that receives a request from the user, processes it updating the session variable which is part of its state, and sends back the response (resp. the HTML generated as output).

Sessions and HTTP requests are modeled as sets of (field, value) pairs. For the resulting HTML we use the modeling approach described in the next section, followed by an abstraction which attempts to retain only the relevant information in the generated page.

[1] We plan to directly generate ASLan models for the model checkers, since this is easier and more efficient. A reverse translation to ASLan++ may be provided to the user.

If the servlet code is loop-free, the entire processing from request to response can be abstracted as a set of transitions with appropriate guards and updates. This corresponds to the "large-block encoding" technique [1, 2] used in model checking with predicate abstraction, and generally ensures more compact and efficiently analyzable models. The most common case involves decisions based on comparing the values of request fields with appropriate (string) constants, represented by abstract constants in the ASLan model.

The code may involve calls to the applications' business logic, including database queries, as in the following example from the bookstore code, where the values of fields `"Login"` and `"Password"` are used to construct an SQL query. Modeling the details of such code is out of the scope of our approach; calls to such external methods are modeled as uninterpreted functions. A mapping back to the source must be maintained to provide the user with an informative interpretation of any attack trace found by the model checker.

```
String sLogin = getParam(request, "Login");
String sPassword = getParam(request, "Password");
java.sql.ResultSet rs = openrs(stat,
   "select␣member_id,␣member_level␣from␣members"
   + "where␣member_login␣=" + toSQL(sLogin, adText)
   + "␣and␣member_password=" + toSQL(sPassword, adText));
   if ( rs.next() ) {   // Login and password passed
     session.setAttribute("UserID", rs.getString(1));
     session.setAttribute("UserRights", rs.getString(2));
    ...
```

In the absence of loops, the resulting ASLan transition system is obtained by accumulating the path conditions, state updates (to the session variable) and the generated output on each path from entry point to one of the exit points of the `_jspService` method that does the server processing.

If the server processing code has loops, a loop cutset needs to be chosen and appropriate state facts added at each cutpoint; then, transitions are generated for all code paths leading directly from one cutpoint to the other.

A sample ASLan++ model manually derived for part of the bookstore application is given in the following. It represents the processing done by three of the JSPs, for login/logout, shopping cart and administration menu. The correct match of username and password is modeled by the fact `isValidCredentials`. Conditions for valid access and two different levels of access rights for an authenticated user are modeled using Horn clauses. The LTL specifications state that a user cannot be recorded as authenticated in a session before having provided valid credentials, and a session cannot have two different authenticated users.

```
specification bookstore
channel_model CCM

entity Environment {
  entity Session(U, S: agent) {
    symbols
      curly: text;
      curlyPass: text;
      nonpublic ses: text.text set; % empty session

    entity Server(Actor, U: agent, Sess: text.text set) {
      symbols
        getUID(text): text;          % SQL queries
        getURights(text): text;
        isValidCredentials(text,text): fact;
        hasAccessToLevel1(text,text): fact;
        hasAccessToLevel2(text,text): fact;
        validU(text.text set, text, text): fact;
        validAccessToLevel1(text.text set, text, text): fact;
        validAccessToLevel2(text.text set, text, text): fact;
        login(text.text set): message;
        shoppingCart(text.text set): message;
        adminMenu(text.text set): message;
        viewLogin: message;
        viewShoppingCart(text): message;
        viewAdminMenu: message;
        Params: text.text set;
        Password: text;
        Remove: text;
        Uname: text;
        UID: text;
        URights: text;
        TRN_UID: text;
        voidSet: text.text set;
        sFormAction: text;
        sLogin: text;
        sLogout: text;
        sPassword: text;
        sUID: text;
        sURights: text;
        sTRN_UID: text;
        sMember: text;
        sFormName: text;
```

```
clauses
  inferValid(Params, Uname, Password):
    validU(Params, Uname, Password) :-
    contains(Params, (sLogin, Uname)) &
    contains(Params, (sPassword, Password)) &
    isValidCredentials(Uname, Password);

  inferHasAccessToLevel1(Sess, UID, URights):
    validAccessToLevel1(Sess, UID, URights) :-
    contains(Sess,(sUID, UID)) &
    contains(Sess, (sURights, URights)) &
    hasAccessToLevel1(UID, URights);

  inferHasAccessToLevel2(Sess, UID, URights):
    validAccessToLevel2(Sess, UID, URights) :-
    contains(Sess,(sUID, UID)) &
    contains(Sess, (sURights, URights)) &
    hasAccessToLevel2(UID, URights);
body {
  isValidCredentials(curly, curlyPass);
  while(true) {
    select {
      on (?U *->* Actor: login(?Params)): {
        select {
          on (Params->contains((sFormName, sLogin))): {
            if (validU(Params, ?Uname, ?Password)) {
              if (Sess->contains((sUID, ?Remove)))
                Sess->remove((sUID, Remove));
              if (Sess->contains((sURights, ?Remove)))
                Sess->remove((sURights, Remove));
              Sess->contains((sUID, getUID(Uname)));
              Sess->contains((sURights, getURights(Uname)));
              U *->* Actor: shoppingCart(Params);
            } else Actor->U: viewLogin;
          }
          on (Params->contains((sFormName, sLogout))): {
            if (Sess->contains((sUID, ?Remove)))
              Sess->remove((sUID, Remove));
            if (Sess->contains((sURights, ?Remove)))
              Sess->remove((sURights, Remove));
            Actor->U: viewLogin;
          }
        }
```

```
            }
            on (U *->* Actor: shoppingCart(?Params)): {
              if (validAccessToLevel1(Sess,?UID,?URights)) {
                if (Params->contains((sFormName,sMember)))
                  if(Sess->contains((sFormAction,?)))
                    U *->* Actor: adminMenu({(sTRN_UID,TRN_UID)});
                  else Actor *->* U: viewShoppingCart(UID);
                else Actor *->* U: viewShoppingCart(UID);
              else U *->* Actor: login(voidSet);
            }
            on (U *->* Actor: adminMenu(?Params)): {
              if(validAccessToLevel2(Sess, ?UID, ?URights))
                Actor *->* U: viewAdminMenu;
              else Actor *->* U: login(voidSet);
            }
          }
        }
        goals
          spec1: forall U1 .
            U (!contains(Sess, (sUID,getUID(U1)))
            & !contains(Sess, (sURights,getURights(U1))),
                                validU(Params,U1,?));
          spec2: forall U1 U2 .
            validU(Params,U1,?) => U (!contains(Sess,(sUID,getUID(U2)))
            & !contains(Sess, (sURights,getURights(U2))),
                                  validU(Params,U2,?)) & U1!=U2;
          spec3: forall UID1 UID2 .
            [] (UID1 != UID2 => !(contains(Sess,(sUID,UID1))
                                & contains(Sess,(sUID,UID2))));
      } % end Server
      body {      % of Session
       new Server(S, i, ses);
      }
    } % end Session
    body { % of Environment
      any A B . Session(A, B) where A != B;
    }
    goals
      auth_user: forall U
        [] (succesfulreq(U) => contains(ses, (sUID.getUID(U)))
           & contains(ses, (sURights.getURights(U))))
} % end Environment
```

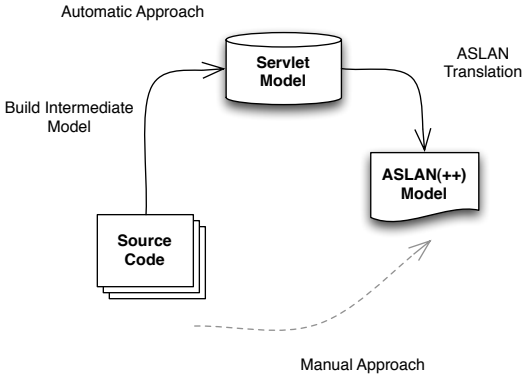

Figure 3: Model extraction process

4.3 Building an Intermediate Model

As discussed above, a complete model of the application has to express both the changes to the application state as a result of an incoming request and the resulting HTML page which is sent back to the client. As a first step, we have implemented a prototype deriving a structural model of all possible web pages that can be generated by an application.

4.3.1 Basic Concepts

Our approach of modeling the responses of a web application is based on the techniques introduced by Offutt and Wu [6] for modeling presentation layers of web applications. The central notion of the approach is the *atomic section*. In essence, an atomic section is an HTML fragment. In a servlet, an atomic section corresponds to a purely sequential block of statements (excluding any form of jumps) that produces an HTML fragment sent to client. Thus, an atomic section is very similar to a basic block.

In general, the response sent back to the client is a composition of atomic sections determined by the state of the client-server interaction, etc. Consequently, for each individual component (e.g., servlet) a transition graph between atomic sections can be generated. This *intra-component* atomic section graph represents all compositions of atomic sections (i.e., full responses) that a component can generate. It is similar to a control-flow graph, except that various paths between two atomic sections without other intervening atomic sections are subsumed in a single transition. Figure 4 shows an example with several atomic sections and their corresponding transitions.

Another kind of transition appears between components and corresponds to links which appear in the response of the server or to actions associated

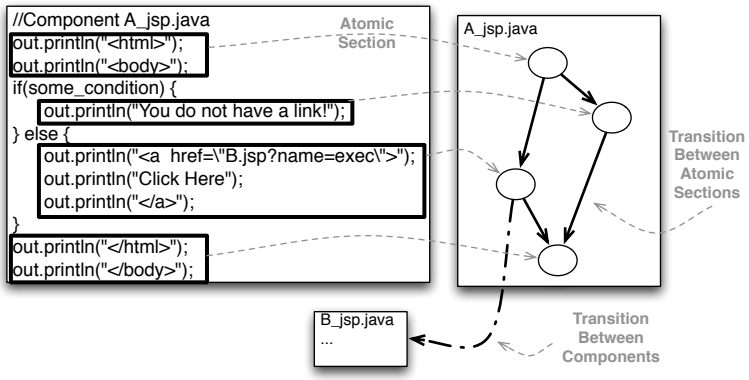

Figure 4: Atomic sections and transitions

to the forms included in the response. Figure 4 also exemplifies a transition of this kind. The components together with the transitions between them form the *inter-component* transition graph.

4.3.2 Meta-Model

In order to build the response model we have defined an initial meta-model to represent information, depicted in Figure 5.

In essence, we represent as first-class entities each of the concepts described in the previous section (i.e., *AtomicSection*, *AtomicSectionTransition*, *ComponentTransition*). Each compilation unit obtained by converting a JSP file into its corresponding servlet is represented by a *Component*. *Form* and *Link* correspond to HTML forms and links and are characterized by their associated parameters when possible. We also represent at this moment the actual HTML produced by an atomic section in the form of *TextFragment* or *UnresolvedFragment*. The latter one also contains expressions that cannot be exactly converted into HTML (e.g., a variable that is not a constant String).

4.3.3 A Prototype Model Builder

We have implemented an early prototype model builder that takes as input the source code of the translated JSPs and produces a model according to the previous meta-model. At present, the builder can construct only the intra-component part of the model (i.e., components, atomic sections, and transitions between them). The main concern is the detection of atomic sections. Currently, such a section is identified as a sequence of invocations to the `print`, `println`, `write` and `writeln` methods on an object of type

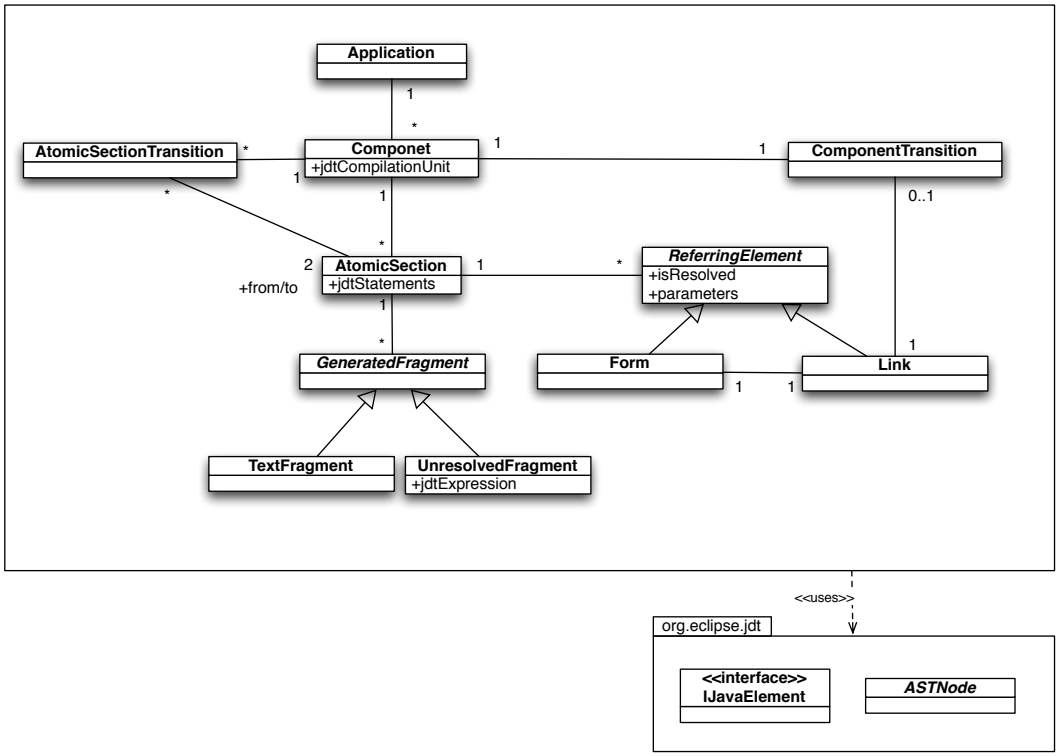

Figure 5: An initial meta-model for an HTTP response

`JspWriter`, which represents the output stream produced by a servlet.

The sequences of such statements are further analyzed. Their arguments are processed, resolved into text (HTML) strings where possible, and concatenated. Parts of the output cannot be resolved as strings, as the arguments to the generating methods are expressions that depend on non-constant values. The atomic section is, therefore, represented as a list of text and "unresolved" fragments, linked together as they show up in the generator sequences. The unresolved fragments maintain the full JDT-specific references in the abstract syntax tree of the program that will allow their subsequent analysis.

The next step dives into the fragments themselves, and extracts the relevant HTML-specific information. For instance, links and forms are extracted, the latter along with the corresponding form actions and input fields. This way, we gather both the information showing the transitions to other possible components (pages), and the sets of parameters exchanged with the clients.

The builder is developed on top of the *org.eclipse.jdt* infrastructure, starting from the components abstract syntax trees. The resulted model is linked to the entities defined in *jdt* (e.g., an atomic sections contains the jdtState-

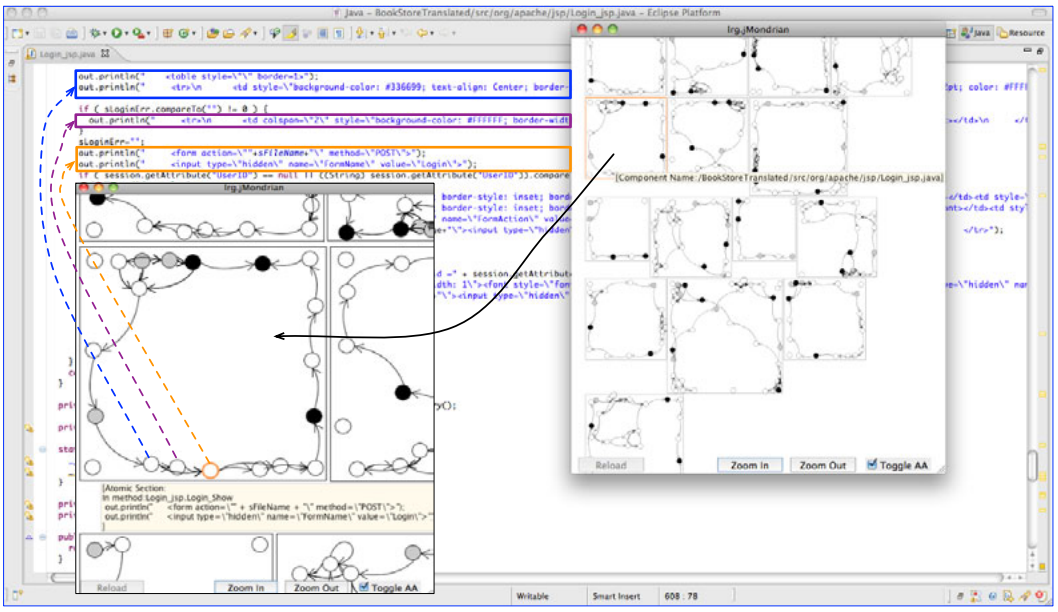

Figure 6: Prototype at work: a visualization of the model

ments that produce it), enabling the navigation back to the AST and, consequently, to the source code.

Figure 6 shows a model obtained using the prototype for the *Login_jsp.java* component of the *GotoCode BookStore* application. A visual representation for all intra-component graphs of the application is shown in the right part of the figure. The left part details the graph for the *Login_jsp.java* component. The source code associated to three of its atomic sections is emphasized.

4.3.4 Challenges and Future Work

The presented model extraction will proceed along two main directions.

First, we will to complete the automatic building process of the model (i.e., to extract the inter-component interactions).

Second, we will enhance this model (currently limited to capturing response structure), and complete the extraction of a behavioral model of the application. This requires extracting from the source code, for each combination of atomic transitions representing a response, two types of information:

- *Guard Condition* – the condition in which that transition is executed, depending on state and input (session information, request parameter values, etc.)

- *State Changes* – the effect of the transition on the state of the system (changes to session information, static variable values, etc.)

This will allow us to implement the second step in the model extraction process of Figure 3, the actual generation of an ASLan/ASLan++ model.

Additionally, a third direction is to try to generalize the model builder. A web application can make use of a large variety of frameworks to create the response for a client (e.g., Struts, Jakarta ECS, etc.). Even worse, these technologies can be combined in an application (e.g., it can use ECS together with pure servlets and pure JSPs). We want to try to find a solution to this problem by incorporating in our builder special hooks, whose role would be to specify the particularities of different frameworks, and thus to decouple the model building process from a particular technology. However, at this moment, it is still open if such a unification is possible.

5 Conclusions

We have selected a set of realistic small to medium size web applications written using Java Server Pages as new case studies for the SPaCIoS project. The security aspects identified in the GotoCode applications match the previous findings of Deliverable D2.1.1 [8]. Due to their structure, these applications are a good starting point for developing and testing an approach for automated model extraction from source code. We have outlined the proposed process and presented first partial results in this direction.

This deliverable will be extended with an analysis of the mOSAIC case study once the relevant security aspects become available. We expect this to happen at month 21 (after an additional 3 months).

References

[1] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *Proceedings of the 9th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 25–32. IEEE, 2009.

[2] A. Gurfinkel, S. Chaki, and S. Sapra. Efficient predicate abstraction of program summaries. In *Proceedings of the Third NASA Formal Methods Symposium*, volume 6617 of *LNCS*, pages 131–145. Springer, 2011.

[3] W. G. Halfond, S. Anand, and A. Orso. Precise interface identification to improve testing and analysis of web applications. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA)*, pages 285–296. ACM, 2009.

[4] W. G. J. Halfond and A. Orso. AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 174–183. ACM, 2005.

[5] W. G. J. Halfond and A. Orso. Improving test case generation for web applications using automated interface discovery. In *Proceedings of the 6th European Software Engineering Conference / ACM Symposium on Foundations of Software Engineering*, pages 145–154. ACM, 2007.

[6] J. Offutt and Y. Wu. Modeling presentation layers of web applications for testing. *Software and System Modeling*, 9(2):257–280, 2010.

[7] OWASP. OWASP WebGoat Project. https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project, 2011.

[8] SPaCIoS. Deliverable 2.1.1: Analysis of the relevant concepts used in the case studies: applicable security concepts, security goals and attack behaviours, 2011.

[9] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated replay and failure detection for web applications. In *Proceedings of the 20th ACM International Conference of Automated Software Engineering (ASE)*, pages 253–262. ACM, 2005.

[10] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Proc. of the 33rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 372–382. ACM, 2006.

[11] Yes Software. Gotocode applications. http://web.archive.org/web/20110430192101/http://gotocode.com/.